JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 93–116 (1999)

Research

Automated Regression Testing Using DBT and *Sleuth*

ANNELIESE VON MAYRHAUSER* and NING ZHANG
Department of Computer Science, Colorado State University, 601 S. Howes Lane, Ft Collins CO 80523, U.S.A.

SUMMARY

Regression testing is an important activity in software maintenance. Current regression testing strategies can be categorized into two groups: 'retest all' and 'selective regression' testing. Each of these two groups encompasses a variety of strategies. In industrial practice, regression testing procedures vary widely. Sometimes, several regression testing techniques are used in combination. Technique selection is also influenced by the expected quality of the system to be tested. Such variations in regression testing strategies and techniques mandate flexibility for a regression testing tool. This paper presents regression testing support for *Sleuth*, a test generation tool based on domain-based testing. We explain the rules for building regression tests for a variety of possible regression testing strategies from retest all strategies to selective regression testing strategies. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: regression testing; black-box testing; domain-based testing; automated test generation; system testing; *Sleuth*

1. INTRODUCTION

As software changes, we have to retest existing functionality to determine whether changes introduced faults into the code. New test cases are needed to test that the modifications work. The process of testing changed software is called *regression testing*. It involves retesting the whole software system or part of it after it is modified, depending on different regression testing strategies. The two basic approaches to regression testing are the *retest all* and the *selective regression testing* strategies. The set of tests run during a regression test is the *regression test suite*. Regression testing can be quite expensive, as, for example, the testing and certification efforts for Year 2000 (Y2K) compliance.

A *retest all* strategy tests the system all over again, in effect assuming that changes could have affected and introduced errors anywhere in the code. A *selective regression testing* strategy assumes that not all parts of the software could have been affected by modifications. Selective regression

*Correspondence to: Dr. A. von Mayrhauser, Department of Computer Science, Colorado State University, 601 S. Howes Lane, Ft Collins, CO 80523, U.S.A. Email: avm@cs.colostate.edu
Contract/grant sponsor: Storage Technology Corporation.
Contract/grant sponsor: Colorado Advanced Software Institute.

CCC 1040–550X/99/020093–24$17.50

Received 4 May 1998
Revised 15 January 1999

testing strategies differ by how they determine which parts of the software could have been affected by changes and how they have to be retested. Regression testing has been used for white-box , gray-box and black-box testing. A large portion of regression testing strategies are white-box strategies. They analyse code changes and their impact, and develop a regression test suite to satisfy white-box test criteria, such as branch or dataflow coverage of the portions of the code that are considered directly or indirectly affected by the code changes (Ostrand and Weyuker, 1988; Karold and Soffa, 1988; Agrawal *et al.*, 1988; Kung *et al.*, 1996; Rothermel and Harrold, 1993, 1994b, 1996; Binkley, 1997).

Fewer researchers have been concerned with gray-box and black-box regression testing (von Mayrhauser *et al.*, 1994a; Leung and White, 1990; Leung, 1995). Here, changes related to the functionality of a system are regression tested. With the advent of more and more systems with a graphical user interface (GUI), it has become important to define regression testing strategies for such systems as well. White (1996) presented a method for regression testing of GUI systems based on a Latin Square design.

While both *retest all* and *selective* regression testing strategies assume that the tester selects from existing tests (as well as develops new ones for new functionality), this does not always meet practical needs. When we interviewed system testers in industry, it became clear that many are concerned that the developer may have fixed the appearance of the bug, but maybe not the bug itself. Merely rerunning the test that found it will not reveal such a problem, additional new tests that probe the same functionality or feature are needed. In such a situation, testers will develop a new test for the same functionality. We call this the *regeneration* strategy to regression testing. In this context, a retest all strategy will develop a whole new set of tests for the entire system while a selective test strategy will develop new tests for the parts of the system affected by the change. Obviously, this is more expensive than reusing existing tests. While always desirable, it becomes even more important to reduce test generation time through automation for such situations.

Automation in the reuse approach focuses on automating ‘optimal’ selection from existing tests (according to some criteria). Automation for the regeneration approach generates test cases automatically for portions of the software affected by change. Our starting point for both is an existing black-box system level test generation tool called *Sleuth*. *Sleuth* is based on domain-based testing (von Mayrhauser *et al.*, 1994b; von Mayrhauser *et al.*, 1994c). Domain-based testing is a test generation method that uses a model of the system under test, the domain model, to generate test cases. The domain model can be customized for a variety of test objectives. These customized test objectives are represented by a testing subdomain. *Sleuth* has been used since 1993 by system testing groups at Storage Technology and elsewhere to test various mass storage devices including tape silos, DASD, etc. It has also been used for generating SQL to test a database, and for telephony testing.

For regression testing support, our approach is to automatically identify domain model changes. The tester has the option of following a selective reuse testing strategy or a regeneration strategy. In the first case, *Sleuth* can be used to select test cases from an existing test suite. When the tester wants to follow a regenerative strategy, *Sleuth* will build a regression subdomain based on domain model changes. *Sleuth* uses these regression test subdomains to generate test cases automatically. Like Leung and White (1991), we are assuming that selective regression testing is reasonable, if the cost of identifying changes and building the regression subdomain is cheaper than the retest all strategy. Another factor is the quality of the code and the changes. We present both a large impact

(assume much is affected by changes) and a small impact (assume little impact) version of the selective regression testing strategy.

Section 2 provides a brief overview of regression testing, in the literature and as we have encountered it in practice. Section 3 provides a short tour of domain-based testing (DBT) and *Sleuth*. Section 4 presents the regression testing rules of DBT via regression subdomain generation. The size of the regression subdomain (i.e., what must be regression tested) depends on the cohesion of elements of the domain. This is discussed next. Section 5 gives examples using the method. Section 6 presents conclusions and points out future work.

2. BACKGROUND

2.1. Regression testing in the literature

Unlike development testing, *regression testing* has the potential to reuse all or part of an existing set of test cases. A 'brute force' approach to regression testing would be to rerun the existing test suite. For large systems, this can be prohibitively expensive. Further, some tests may no longer be valid. For example, when data formats change as in expanding a two-digit year to a four-digit year (the Y2K problem), existing tests using two-digit year data may not make sense any more and cause either error messages from the system or outright system failure, but not the expected outputs associated with the old test case. In such cases, the tester has to determine whether discrepancies between expected and actual output are due to a test case that is no longer applicable or due to a failure. This further increases the cost for this *retest all* approach.

A regression testing strategy that selects a subset of the original test suite according to some criteria (e.g., whether a test is still applicable, or whether it activates portions of the code that were changed) is a *selective regression test* strategy. The objective of a selective regression test strategy is to reduce the time and effort of regression testing while preserving the efficacy of the test suite in revealing faults.

For either strategy, the tester may need to develop new tests to exercise new features of the software or to cover parts of the software that are no longer covered by existing tests. Rothermel and Harrold (1996) provided a definition of regression testing as consisting of the following steps:

1. Identify changes.
2. Determine which of the currently existing test cases will remain valid for the new version of the software (eliminate all tests that are no longer applicable—this results in a set of tests T', a subset of the original test suite T).
3. Test the modified software with T'.
4. If necessary, test parts of the software that are not tested adequately with T' by generating new test cases T''. What it means to test adequately depends on the test criterion chosen.
5. Execute the modified software with T''.

Steps 2 and 3 test whether the modifications have broken any existing functions. Steps 4 and 5 test whether the modifications work.

T' is determined based on the classification of existing test cases and the regression testing approach, retest all or selective regression testing. Leung and White (1989) classified the existing

test cases as *reusable*, *retestable* and *obsolete*. A reusable test case does not test the software modification and should produce the same results as previous tests. It need not be rerun. A retestable test case tests the software modification and must be rerun. An obsolete test case no longer applies to the modified software. Obsolete tests can be removed from the regression test suite. All reuse-based selective regression testing techniques select retestable test cases from the existing test suite.

Regeneration-based regression testing focuses on how to effectively and efficiently generate a new test suite. A regeneration-based *retest all* approach generates a complete, new test suite for the software. This approach may duplicate effort on the portion of the code or functionality which has not changed or has not been affected by changes. However, according to the cost model of Leung and White (1991), a regeneration-based retest all approach is a more reasonable solution if the cost of reuse is higher than the effort saved by reuse. A regeneration-based retest all approach is equivalent to *development testing*. Regeneration-based *selective regression testing* by contrast, attempts to reduce the effort to retest modified software by limiting the testing scope to the portions of the software that are affected by the changes.

Regression testing techniques differ depending on focus and objectives. Table 1 shows a sampling of techniques in various categories. These include white-box versus black-box testing and testing of procedural languages rather than object-orientated systems. Most techniques focus on reuse-based selective regression testing. Leung and White (1989) analysed the fundamental issues involved in classifying test cases for regression testing. Control flow graph changes are used as an example to classify existing test cases. Agrawal *et al.* (1988) used slices to select retestable test cases. Rothermel and Harrold (1993) formalized a safe and efficient algorithm for test case selection based on changes to the control dependency graph. Harrold and Soffa (1988), and Ostrand and Weyuker (1988) analysed programs written in procedural languages based on data flow graphs (DFG), and defined reuse-based regression testing rules based on DFG changes. Rothermel and Harrold (1994b), Hsia *et al.* (1997) and Kung *et al.* (1996) extended regression test selection techniques to object-oriented programs. Rothermel and Harrold (1994b) based their selective regression testing for OO software on an interprocedural program dependence graph (IPDG) instead of the object relation graph (ORG) (Hsia *et al.*, 1997; Kung *et al.*, 1996). Selective retesting based on an IPDG extends the scope of changes (in this case class changes) to derived classes as well as programs that use modified classes.

Selective regression test strategies should be both efficient and effective. Rosenblum and Weyuker (1996) investigated the efficiency of different white-box regression testing techniques. Leung (1995) discussed the effectiveness of selective white-box regression testing. Leung and White (1989) and Rothermel and Harrold (1994a, 1996) provided a framework for regression testing concepts and their evaluation with regards to efficiency and effectiveness.

Further improvements in the efficiency of a regression testing strategy are possible, if selection of retestable tests can be automated. Hartmann and Robson (1988), Harrold and Soffa (1988), Chen, Rosenblum and Vo (1994), and White and Narayanswamy (1993) focused on the automation of white-box regression testing. Leung and White (1990), and von Mayrhauser *et al.* (1994a) provided black-box regression testing methods with various degrees of automation.

Most of the research has been in the area of white-box testing. This paper focuses on the automation of regression testing for black-box testing (shown as stars in Table 1). Specifically, we base our regression testing approach on domain-based testing (von Mayrhauser *et al.*, 1994c).

Table 1. Regression testing

Regression testing techniques	Concept framework	Procedural language-based program	OO program	Efficiency	Effectiveness	Automation and tools
White-box testing (code-based)	[1], [2]	[3], [4], [5]	[6], [7], [8]	[9]	[10]	[11], [12], [13], [14]
Black-box testing (specification-based)	[15], [16]					***

[1] Rothermel and Harrold (1996)
[2] Leung and White (1989)
[3] Ostrand and Weyuker (1988)
[4] Rothermel and Harrold (1993)
[5] Agrawal *et al.* (1988)
[6] Hsia *et al.* (1997)
[7] Kung *et al.* (1996)
[8] Rothermel and Harrold (1994b)
[9] Rosenblum and Weyuker (1996)
[10] Leung (1995)
[11] Chen, Rosenblum and Vo (1994)
[12] Harrold and Soffa (1988)
[13] Hartman and Robson (1988)
[14] White and Narayanswamy (1993)
[15] von Mayrhauser *et al.* (1994a)
[16] Leung and White (1990)

2.2. Regression testing in practice

Regression testing occupies a more and more important role in industry, because of evolutionary development, successive releases of new versions, as well as standard maintenance. Besides the regression testing needs caused by software changes due to evolutionary development, successive releases and maintenance, deadline constraints, the nature of the changes and confidence in the quality of the prior version of the software are all factors that influence which (*ad hoc*) regression testing strategies are used.

Table 2 describes types of situations that may call for different regression testing approaches. We distinguish between four different approaches to regression testing. Table 3 shows their corresponding strategies. They range from selecting the smallest possible (based on selection rules) set of tests to developing a complete set of tests to test the entire system from scratch. When the new release is based on major specification changes and functionality enhancements, a full test cycle is necessary. Testers have to design the test plan and generate new test cases for the entire system. This is the regeneration-based retest all approach. In such a situation, there is no difference between development and regression testing. What is considered ‘major change’ can vary greatly between organizations and may depend on the pervasiveness of changes, the proportion of obsolete tests, as well as the expected quality of the software or the existing test suite. For example, a system that has undergone extensive Y2K updates may require new input data formats for much of its input. Such a

Table 2. Regression testing in various situations

Approach	Situation descriptors			
	Confidence in software quality	Change nature	Change impact	Schedule constraints
Full new test cycle	Low	Key code	Extensive	None
Minimalistic regression test	High	Isolated	Localized	Very tight
Expanded scope of regression test	Moderate	Key code	Extensive	Moderate
Full reuse of existing test suite	Low	Series of fixes	Extensive	Moderate

Table 3. Regression testing approaches

Approaches	Strategies
(1) Full new test cycle	• redesign entire test plan • rebuild the whole test suite
(2) Minimum regression test	• reuse test plan as much as possible • rerun minimum number of retestable test cases • generate minimum number of new test cases for changes
(3) Expanded scope of regression testing	• reuse part of test plan • rerun all retestable test cases • generate new test cases on the full scope of changes
(4) Full reuse of existing test suite	• reuse the test plan and test suite

system may best be tested as if it were new, particularly when its quality or the quality of its existing test suite is a concern. Obviously, one would like to avoid this situation as it represents the most expensive option.

At the other end of the spectrum is the 'minimalistic' test suite. In this context minimalistic is used with regards to a practicable set of selection rules; mathematically minimal test selection is NP-complete. Thus, our concept of minimalistic may include techniques that could under some circumstances cause false positives and false negatives. In this case we select the smallest set of retestable tests we are able to identify that which will still test modified portions of the code. This minimalist approach is used when the market window is small, or when the changes fixed defects or added features that must be released in a hurry. Testers reuse the original test plan, and define the scope of changes as small as possible in order to keep regression testing to a minimum. Then they select the fewest possible retestable test cases from an existing test suite. They also generate the fewest possible number of new test cases to test new functions. This approach is a combination of minimalistic reuse-based and regeneration-based selective retest. How well this works depends on the quality of the old version, the changes and how localized the changes were.

When testers do not have enough confidence that changes are localized, testers have to enlarge the testing scope to ensure all affected parts are tested. This approach reuses part of the existing test plan and test suite, but generates new test cases on the full scope of changes. It may also rerun a larger number of previous test cases based on defining a bigger portion of the software as affected

by the change (to test that all existing features still work well). It is an extended version of selective retest.

Finally, if the changes did not involve any specification changes, testers may rerun the whole test suite again. This is the reuse-based retest all approach and can be expensive, as pointed out earlier.

Applying multiple testing techniques is common when testing software. When testing a modified software product, testers may want to apply more than one regression testing strategy to increase their confidence in the changed software. Different testers may choose different regression testing techniques or strategies based on their own experience and judgement. To reduce effort and costs, testing tools are used. However, to support multiple regression testing strategies, such tools should be general and flexible. They should be able to support different regression testing requests to achieve different regression testing objectives. This is one reason why we used *Domain-based testing* and *Sleuth* as the basis for defining regression testing support. We explain both next.

3. DOMAIN-BASED TESTING AND *SLEUTH*

3.1. Example domain: StorageTek robot tape library command language

We illustrate *domain-based testing* and *Sleuth* on a large system (over 1 million lines of code). Storage Technology Corporation (StorageTek) produces an automated cartridge system (ACS) that stores and retrieves cartridge tapes. The system maintains magnetic tape cartridges in a 12-sided 'silo' called a library storage module (LSM). Each LSM contains a vision-assisted robot and storage for up to 6 000 cartridges. Tapes occupy cells in the panels. New tapes are entered through a special door called a cartridge access port (CAP). Figure 1 shows a single LSM with tape drives, access port and control unit.

The ACS and its components are controlled through a command language interface called the *host software component* (HSC). Each HSC supports from 1–16 ACS systems. HSC commands manipulate cartridges, set the status of various components in the system and display status information to the operator's console. The domain models used in our regression test case study represent three different HSC versions (1.0.2, 2.0.0 and 2.0.1). HSC version 1.0.2 has 32 commands and 32 parameters. Versions 2.0.0 and 2.0.1 have 31 commands and 39 parameters each.

3.2. Domain-based testing

3.2.1. Overview

Domain-based testing (DBT) is a test generation method based on two concepts from software reuse, domain analysis and domain modelling. In DBT, the *domain* is an application domain of the software which is to be tested (von Mayrhauser and Crawford-Hines, 1993). Domain analysis (Debaud, Moopen and Rugaber, 1994) refers to the process of analyzing an application domain for reusability. Specifically for DBT, domain analysis is concerned with capturing application domain concepts in a manner that facilitates the test generation as well as identification of test cases that can be cost-effectively reused. *Domain analysis* (von Mayrhauser *et al.*, 1994c) extracts

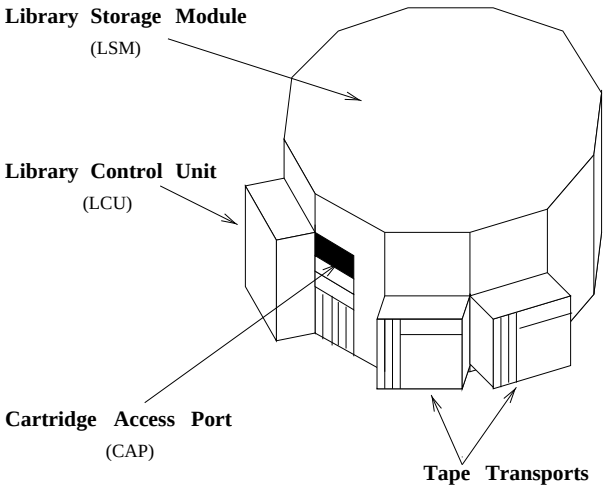

Figure 1. Library storage module

common information about a problem domain, specifies the operations of the domain and packages this information. The result of a domain analysis is called a *domain model*. When used during development, domain models represent the application domain and serve as a mechanism to create instances of reusable components. When used for testing, a domain model need only describe aspects of domain objects and their interactions that provide enough information to generate syntactically and semantically meaningful tests. The domain model only describes information relevant for test generation rather than development. Modifications to the domain model create testing subdomains that describe parts of the software that should be tested, or focus on error recovery by providing a testing subdomain that describes erroneous behaviour for the domain. Table 4 lists the steps for the domain analysis and the domain model components generated at each step. The next subsections describe details of applying this analysis (von Mayrhauser *et al.*, 1994c). We use the StorageTek automated tape library to illustrate each step.

3.2.2. Object analysis

The first step identifies the *objects* of the system, *object elements* and *relationships* between the objects. *Objects* denote physical or logical entities from the problem domain. *Object elements* define qualities and properties of the object. Object relationships are used to define parameter value constraints. The domain analyst uses *syntactic elements*, *user documentation* and *user semantic interpretation* for object identification.

Step 1 identifies objects and object relationships of the system under test. Each parameter in the command language is associated with an object. Figure 2 shows how two HSC commands from the robot tape library have parameters that relate to three domain objects `Cartridge`, `Tape Transport` and `HSC`. The parameters within each object are called *object elements*. Object elements are similar to the concept of *object attributes* in OOA/OOD. The *object glossary* records objects and their object elements. Table 5 shows the `LSM` entry from the object glossary.

Table 4. Domain analysis steps for domain based testing (von Mayrhauser *et al.*, 1994c)

Domain analysis step	Domain model component
1. Object analysis	
1.1. Define objects and object elements	Set of objects
1.2. Define default parameter values and object/object element glossaries	Default parameter value sets
1.3. Define object hierarchy	Object hierarchy
1.4. Annotate hierarchy with parameter constraints	Parameter constraint rules
2. Command definition	
2.1. Command language representation	Command language syntax
2.2. Identify intracommand rules	Intracommand rules
3. Script definition (command sequencing)	
3.1. Script class definition	Script classes
3.2. Script rule definition	Command sequencing rules

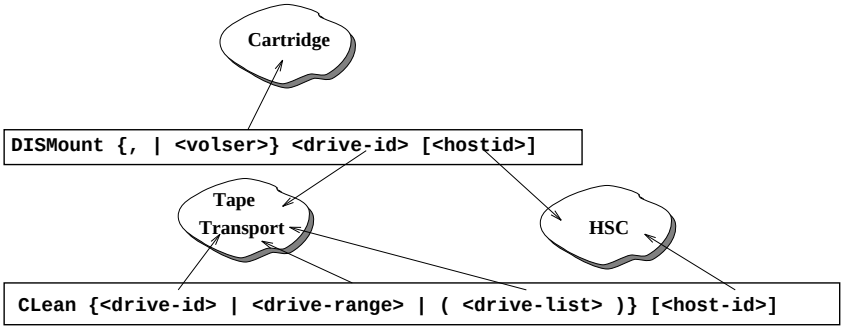

Figure 2. Analysing HSC commands for objects and object elements (von Mayrhauser et al., 1994c)

Table 5. Object glossary entry for the `LSM` object (von Mayrhauser *et al.*, 1994c)

Object	`LSM`
Description	Library storage module—a single tape 'silo'
Parameters	*lsm-id, lsm-ls, lsm-rg*

Next, we define valid parameter values for each object element. This is recorded in the *object element glossary*. To select parameter values, automated test generation must know the range of values for each element and its representation (single value, range or list of values). Table 6 shows an entry from the object element glossary for the StorageTek HSC command language.

The next step in the domain analysis determines relationships between objects. These relationships are captured in an *object hierarchy* in the form of a 'part of' hierarchy. The object hierarchy is annotated with parameter constraint rules which describe how the choice of one parameter value constrains the choices for another. For example (see Figure 3), each ACS supports

Table 6. LSM entry from the HSC object element glossary (von Mayrhauser *et al.*, 1994c)

Parameter	*lsm*
Full name	Library storage module (LSM) identifier
Definition	Names an instance of an LSM within an ACS
Values	000. . . FFF
Object	`LSM`
Representation	*lsm-id, lsm-ls, lsm-rg*

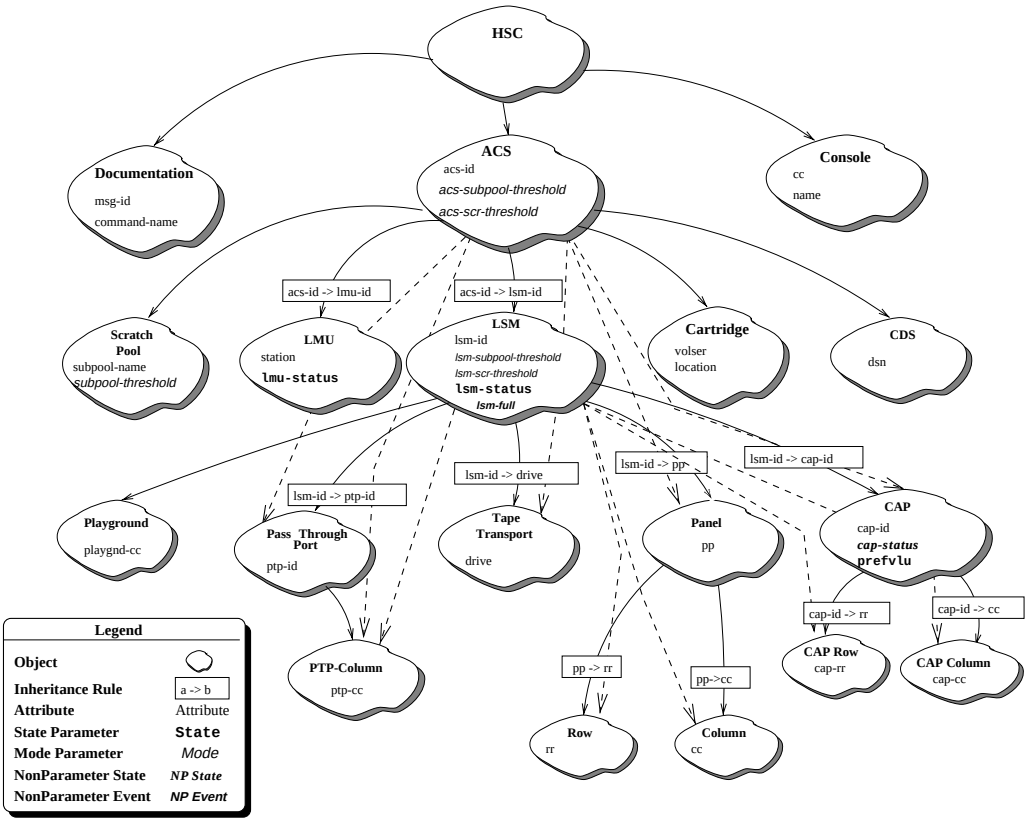

Figure 3. StorageTek object hierarchy (von Mayrhauser et al., 1994c)

up to 16 LSMs (shown in the figure as an arrow from the ACS object to the LSM object). Each LSM contains panels, tape drives, cartridge access ports, etc. Arrows from the LSM to each object denote this structure. Annotations on the arcs state parameter constraint rules. For instance, the choice of an ACS (i.e., a specific *acs-id* value) constrains choices for the LSM (i.e., possible *lsm-id* values).

Table 7. Script rule: parameter value selection

Rule	Description
p^*	Choose any valid value for p
p	Choose a previously bound value for p
p-	Choose any except a previously bound value for p

3.2.3. *Command definition*

Step 2 of the domain analysis defines command syntax and intracommand rules for each command. These rules identify constraints placed on parameter value selection within a command. For example, user documentation for the HSC `MOVE` command require that 'when moving a tape within the same LSM, the source and destination panels must be different'. The domain model captures this intracommand rule as: `if (lsm$1 = lsm$2) => (panel$1 ≠ panel$2)`.

3.2.4. *Script definition*

Step 3 describes dynamic system behaviour by capturing rules for sequencing commands and by classifying commands from the problem domain. Sequencing information is necessary because arbitrarily ordering a list of commands rarely produces semantically correct test cases.

The first part of scripting analysis is to group related commands into *scripting classes*. Scripting classes can partition by function, object or object element. Functional partitioning creates scripting classes that include commands that perform similar actions. For example, in the StorageTek domain, the *set-up* class includes all commands that perform system set-up functions; the *action* class includes commands that manipulate and exercise the robot tape library. If we partition the commands by object, then we examine each object and create a class that contains all commands that influence that object.

Next, we define command sequencing rules. This includes both command sequencing information (i.e., script rules) and parameter binding information for each rule. For example, in the robot tape library, tapes must be mounted before they can be dismounted. These rules have the power of regular expressions. They can contain both commands and scripting class names.

A script rule can be annotated with parameter binding rules, as shown in Table 7. The first rule, p^*, states that the value for parameter p can be selected from any valid choice as long as it fulfils parameter constraint rules. The second rule, p, restricts the value of parameter p to a previously bound value. The third rule, p-, denotes that parameter p can be selected from any valid choice except for the currently bound value of p. To illustrate, the `MOUNT -> DISMOUNT` sequence is annotated with script parameter selection rules.

```
MOUNT tape-id* drive-id*
Any+
DISMOUNT tape-id drive-id
```

This rule states that the *tape-id* and *drive-id* parameters can be selected from any valid choice for the `MOUNT` command while the `DISMOUNT` command must use the previously bound value

for the *tape-id* and the *drive-id* parameters: the tape should be dismounted from the drive where it was mounted. Additionally, there should be at least one command between the `MOUNT` and the `DISMOUNT`.

3.3. Test subdomain definition

The test subdomain couples software testing strategies with the domain model representation. The tester defines the test criteria by altering, modifying and configuring the domain model. Any modification to a domain model is called a *test subdomain*.

The *test subdomain* represents a customized domain model that focuses test case generation on those parts of the software that need to be tested. A test subdomain may be a subset or a superset of the original domain model. A subset restricts the parameters and commands generated in a test case and a superset allows greater freedom in test generation by turning semantic rules off (script rules, intracommand rules, parameter inheritance rules).

Test subdomain definition can be at the script class level. Script classes define sets of commands with similar functionality. Restricting the set of commands in a script class forces test generation to eliminate particular commands from the test case and thus, focuses test generation on a subset of the application domain. For example (see Figure 2), when testing tape transport related commands, only those commands that use *drive-id*, *drive-ls* and *drive-rg* as parameters need to be tested. Thus, the test subdomain need only contain commands `CLean`, `DISMount`, `Mount`, `View`. Compared with the 32 commands in the full domain model this is quite a small subset, which should require less test effort.

3.4. *Sleuth*: an automated test generation tool

Sleuth is an automated test generation tool developed at Colorado State University. It supports domain-based testing (von Mayrhauser *et al.*, 1994b) by providing tools and utilities for test generation. It is based on the test generation process model (shown in Figure 4). The first step in this process is the domain analysis described above. *Sleuth* provides domain capture tools for this (under the toolbar *Specification* in Figure 5). The domain model, D_0^v, captures the syntax and semantics of the system under test. Tests generated from D_0^v are 'valid' sequences of commands that follow all syntax and semantic rules in the domain model. The resulting domain model can be customized into a test subdomain (under the toolbar *Configuration* in Figure 5). Often, the domain model is modified to test a specific system configuration or to test a particular feature of the system under test. This creates a *test subdomain*, TSD_j^v, one for each test objective j. *Test criteria* influence the test subdomain definition and the test generation steps. Test engineers use their knowledge to modify the domain model. They also guide test generation by recalling archived test suites, identifying how many commands to generate, and what commands to generate.

Test generation uses the test subdomain and instructions from the test engineer to create test suites, T_j^v. A *test suite* for DBT may contain *test scripts*, *test templates* and *test cases*. Test scripts are lists of command names (cf. panel marked *Scripting* in Figure 5). A command template is a list of commands with place holders for parameters (cf. panel marked *Commands* in Figure 5). A test case is a list of fully parametrized commands conforming to the syntax and semantics of the software under test (cf. panel marked *Parameters* in Figure 5).

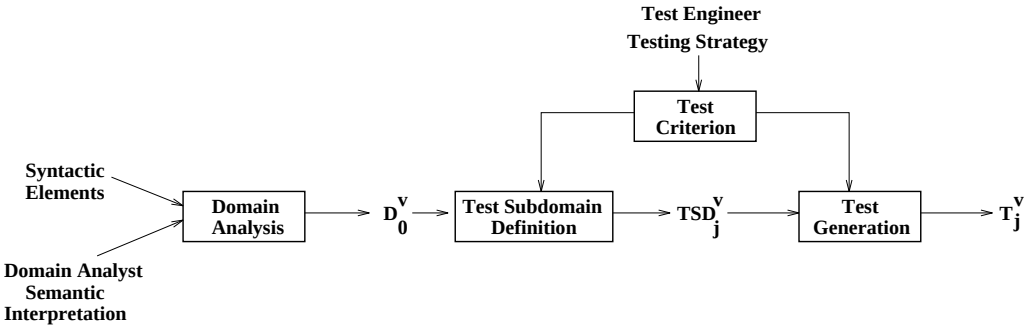

Figure 4. Test generation process model

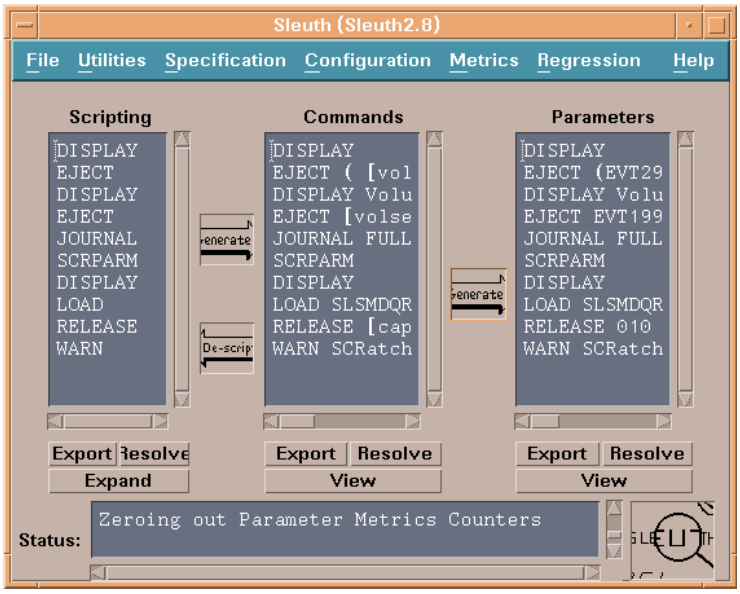

Figure 5. Sleuth *main window*

Corresponding to the three levels of abstraction for test suites, *Sleuth* uses a three-stage test generation process. In the first stage, script classes and script rules are expanded. This produces a list of command names. The second stage creates a command template. The last stage uses script parameter binding rules, intracommand rules, parameter value sets and parameter constraint rules to create a fully parametrized list of executable commands. Test suites may be saved and reused at any of the levels. Regression testing rules can select retestable test suites from these archived tests for implementation of a selective, reuse-based strategy. On the other hand, rules for definition of a regression testing subdomain builds a subdomain of the software under test that can form the basis for a selective regeneration-based regression test.

4. REGRESSION TESTING

4.1. Identify regression testing needs

Different regression testing needs lead to different regression testing strategies. Regression testing strategies differ in how they reuse subset T' of the existing test suite T, and in how the new test set T'' is created.

We categorize four different objectives:

1. Retest all.
   All existing tests are rerun, no new ones are developed.
2. Retest minimalist existing test suite and create a new minimalist test suite for testing modifications.
   Minimalist for T' is based on defining scope of impact of changes as the smallest possible based on changes made to the domain model and then eliminating duplicate tests from the set of retestable tests that cover the defined change impact. Similarly, T'' consists of the smallest number of test cases that test new code. Either may be too small to really cover the true impact of change (i.e., we are not testing enough). This means they are not safe.
3. Safely retest the existing test suite and create a new safe test suite for testing modifications.
   To deal with the problem of unsafe regression testing, one may define the impact of change more conservatively, assuming a larger impact of changes (i.e., it is guaranteed that all parts affected by change are included in what is to be regression tested. Imprecise but safe techniques thus may define impact of change bigger than it is). This leads to a larger set of retestable test cases T'. T' may not be the smallest set possible (i.e., removal of coverage duplicates in terms of regression subdomain coverage). Analogously, T'' may be larger than in the minimalist approach based on more demanding, but safer test criteria.
4. Retest none, but create a totally new test suite.
   In this last category, all tests are developed from scratch. While one may reuse a higher level test plan, actual tests will differ from the set T of original test cases.

Any of those strategies is reasonable under certain circumstances. Factors that affect which strategy to select include quality of prior version of software, nature (enhancement, corrections, etc.) and extent of change, expected quality of changes and quality of existing test cases as discussed earlier.

In *Sleuth* and DBT, we provide support for all four regression testing strategies. Regeneration strategies are based on defining the extent of the change by its associated regression subdomain (i.e., the parts of the domain model that must be regression tested). Regression test suites are then generated based on this regression subdomain. *Small impact* versus *large impact* strategies define the impact of change (and the associated regression subdomain) based on assuming minimal impact versus including indirect effect of changes as well. We explain regeneration strategies in Section 4.2. DBT and *Sleuth* also support reuse strategies. These are described in Section 4.3.

4.2. Regeneration strategies

The key to the regeneration strategies for regression testing in DBT is the definition of the regression subdomain. Rules for building it are based on the type of change to an element of

Table 8. Sets used for determining regression test subdomain

Cmd	Set of commands directly affected by change
Param	Set of parameters directly affected by change
Factor	Set of parameters indirectly affected by change

the domain model (commands, parameters, various rules). The approach described below is an extension of von Mayrhauser *et al.* (1994a). That approach only considered adding to and deleting commands from the domain model, but did not address modification to commands, nor changes to rules and parameters.

The extent of the change is reflected in the content of the *Regression Subdomain*. It is constructed by including all parts of the Domain Model that are affected by the changes. When determining the impact of such changes, we consider two major categories: *Commands Affected by Changes (Cmd)* and *Parameters Affected by Changes (Param)*. The latter reflects direct impact of change (i.e., parameters that occur in changed commands and parameters whose values or parameter inheritance rules have changed). We then construct the regression subdomain by including all commands in *Cmd* and all commands that have parameters in *Param* (as well as any scripting, intracommand and inheritance rules related to these commands and parameters). This definition of a regression subdomain only includes *direct* impact of change: we call this a *small impact* regression testing strategy, as it builds the smallest consistent regression subdomain.

Let us consider next a more conservative approach to building the regression subdomain that includes consideration of *indirect* change impact. We call this the *large impact* regression testing strategy. When commands share parameters, we may see indirect change impact on commands that, while not changed themselves, share some parameters with a changed command. Commands may relate to each other via the parameters on which they operate. For example, in Figure 2 the commands `CLean` and `DISMount` share two parameters `drive-id` and `host-id`. Relationships between commands through shared parameters is an indication of cohesion in the command language. When changes are made, these relationships need to be taken into account when building a regression subdomain. They reflect indirect impact of a change. Including indirect change impact in defining the regression test subdomain characterizes the large impact regression test strategy. The set of parameters that are indirectly affected by changes is called *Factor*.

Thus, if a command is in *Cmd* or at least one of its parameters is in *Param*, the command is directly affected by changes. If, on the other hand, a command is not in *Cmd*, none of its parameters is in *Param*, but at least one of its parameters is in *Factor*, then the command is indirectly affected by changes. Table 8 shows the definitions of the three sets *Cmd*, *Param* and *Factor*. Table 9 shows how to construct *Cmd*, *Param* and *Factor*. The algorithm *exact* takes a command name as input and returns a set of parameters which are used in the command. The algorithm *valid_command* determines whether the command is still in the modified domain model. The algorithm differs depending on the type of change operation. Generally, we have three types of changes: add, delete and modify. These change operations can be applied to commands, scripting rules, intracommand rules, and parameter values or inheritance rules. The table shows how to build the regression subdomain for both *small impact* and *large impact* regression testing. For

Table 9. Domain model changes

Domain model changes	Building small impact regression subdomain	Building large impact regression subdomain
Command syntax changes		
• Add/modify commands (c)	$Cmd = Cmd \cup \{c\}$;	$Factor = Factor \cup extract(c)$;
• Delete commands (c)	for every $p \in extract(c)$ { if ($valid\_param(p)$) $Param = Param \cup \{p\}$; }	
Scripting rule changes		
• Add new scripting rules (S_Rule(c_1, c_2))	$Cmd = Cmd \cup \{c_1, c_2\}$;	$Factor = Factor \cup extract(c_1) \cup extract(c_2)$;

• Delete scripting rules (S_Rule(c_1, c_2))	if ($valid\_command(c_1)$)
	$Cmd = Cmd \cup \{c_1\}$;
	if ($valid\_command(c_2)$)
	$Cmd = Cmd \cup \{c_2\}$;
		if ($valid\_command(c_1)$)
		$Factor = Factor \cup extract(c_1)$;
		if ($valid\_command(c_2)$)
		$Factor = Factor \cup extract(c_2)$;

• Modify scripting rules (S_Rule(c_1, c_2)) from (S_Rule(c_1', c_2'))	$Cmd = Cmd \cup \{c_1, c_2\}$;
	if ($valid\_command(c_1')$)
	$Cmd = Cmd \cup \{c_1'\}$;
	if ($valid\_command(c_2')$)
	$Cmd = Cmd \cup \{c_2'\}$;
		$Factor = Factor \cup extract(c_1) \cup extract(c_2)$;
		if ($valid\_command(c_1')$)
		$Factor = Factor \cup extract(c_1')$;
		if ($valid\_command(c_2')$)
		$Factor = Factor \cup extract(c_2')$;

Intracommand rule changes		
• Add/modify intracommand rule for command (c)	$Cmd = Cmd \cup \{c\}$;	$Factor = Factor \cup extract(c)$;
• Delete intracommand rule for command (c)	if ($valid\_command(c)$) $Cmd = Cmd \cup \{c\}$;	if !($valid\_command(c)$) $Factor = Factor \cup extract(c)$;
• Changed intracommand rule (c)	$Cmd = Cmd \cup \{c\}$;	$Factor = Factor \cup extract(c)$;
Parameter value changes/ inheritance rule changes (p)	$Param = Param \cup \{p\}$;	

example, when adding a new or modifying an existing command, the small impact strategy only includes the new command in the regression testing domain, but not commands with which it shares parameters. The large impact regression subdomain achieves that by extracting all parameters in the new command and adding them to the *Factor* set, thereby including command interactions via shared parameters. Sometimes there is no difference between the small impact and large impact approach. For example, when deleting a command, all its parameters are directly affected by the change. None are considered indirectly affected.

Based on these two regression testing approaches, small impact or large impact, two different regression subdomains are built, the *basic* and the *extended* regression subdomains. The basic regression subdomain consists of,

$$C = Cmd \cup \{c | c \text{ has at least one parameter } \in Param\}$$

That is, only commands that are directly impacted by domain model changes are included. The extended subdomain is defined as,

$$C = Cmd \cup \{c | c \text{ has at least one parameter } \in Param\} \cup \{c | c \text{ has at least one parameter } \in Factor\}$$

Thus, the extended regression subdomain includes all commands that are directly or indirectly affected by changes through sharing parameters with directly affected commands (not necessarily changed ones). This regression subdomain is now used by the testers in the same way as any other testing subdomain. They can use it in *Sleuth* and ask *Sleuth* to generate a series of test cases, they can disable some of the scripting rules and generate semantically incorrect behaviour as part of error recovery testing, etc.

4.3. Reuse strategies

Existing test case classification is the key to selective reuse strategies. According to domain model changes, existing test cases can be categorized as reusable, retestable or obsolete (von Mayrhauser *et al.*, 1994a). In DBT and *Sleuth*, there is another dimension for test case classification, *functional* and *error recovery*. Subdomains can be created from a given domain model by turning rules on or off or by modifying them. When some semantic rules are turned off or modified, the test cases generated are semantically wrong. They can be used to test error recovery of the system under test. Such test cases are called *error recovery* test cases. Test cases that test valid system operation are called *functional*.

To categorize existing test cases for DBT and *Sleuth*, we consider both functional and error recovery test cases. Functional test cases test regular system operation. Error recovery test cases test invalid system use (error recovery). Thus, we have two sets of existing test cases (functional and error recovery) for selection. Based on the type of domain model change, originally valid functional test cases can become obsolete in the new domain model or vice versa. For example, when a newly added sequencing rule is added (e.g., `MOUnt * DISMount`) for command `MOUnt` in the modified domain model, the originally valid functional test case (e.g., `MOUnt CLean DISPlay`) is not valid any more. In Table 10, we summarize the possible transformation of *modification-traversing* test cases based on the kinds of domain model changes. Modification traversing is based on comparing the old domain model with the modified domain model. For example, when the command syntax changes, previously valid functional tests (column 2) related to this command become error recovery tests (they do not conform to the new syntax entry in column 2). Thus, they test the error recovery capability of the system with regards to syntactically erroneous commands. In the table, the new category of an existing test case is specified in two dimensions. One is whether it is retestable, reusable or obsolete, the other is whether it is functional or tests error recovery. This analysis is automatable and provides testers with support in selecting retestable test cases from the *Sleuth* archives.

4.4. Impact of domain cohesion on regression subdomain

What causes differences between the basic and the extended regression subdomain? Essentially, the degree to which parameters are shared between commands. This is what we call *Domain*

Table 10. Transformation of affected test cases

Domain model changes	Functional tests	Error recovery tests
Command syntax changes		
• Addition	N/A	N/A
• Deletion	Obsolete	N/A
• Change	Obsolete Functional/reusable	N/A
Parameter value changes		
• Addition	N/A	N/A
• Deletion	Obsolete	Obsolete
• Change	Functional/reusable/ error recovery/retestable	functional/retestable/ error recovery/reusable
Inheritance rule changes		
• Addition	Error recovery/retestable/ functional/retestable	N/A N/A
• Deletion	Functional/reusable	Functional/retestable
• Change	Error recovery/retestable/ functional/reusable	Functional/retestable/ error recovery/reusable
Scripting rule changes		
• Addition	Error recovery/retestable/ functional/retestable	N/A
• Deletion	Functional/retestable	Functional/retestable
• Change	Functional/reusable/ error recovery/retestable	Error recovery/reusable/ functional/retestable
Intracommand rule changes		
• Addition	Error recovery/retestable/ functional/retestable	N/A
• Deletion	Functional/retestable	Functional/retestable
• Change	Functional/reusable/ error recovery/retestable	Functional/retestable/ error recovery/reusable

Cohesion. When actions operate on many shared objects, commands share many parameters and the domain has high cohesion. Because of how a regression subdomain is built, such high cohesion may result in the extended regression subdomain extending to the whole domain model. This implies a retest all strategy. Thus, cohesion in a command language determines the extent of the regression testing effort. This implies that designing a language with a preference for actions on objects, and limited sharing of objects or object attributes across subsets of commands will result in smaller regression domains and less regression test effort.

5. EXAMPLES

To demonstrate the method in practice, we chose StorageTek's product, the Automated Cartridge System (ACS) and its Host Software Component (HSC), as our testing domain. HSC commands manipulate cartridges, set the status of various components in the system, and display status information to the operator's console. We built domain models for three HSC versions (1.0.2, 2.0.0 and 2.0.1).

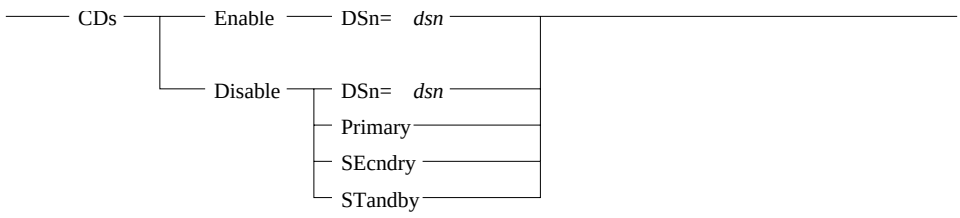

Figure 6. CDs syntax in HSC v1.0.2

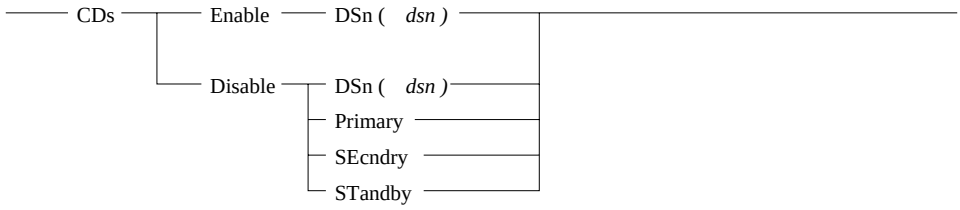

Figure 7. CDs syntax in HSC v2.0.0

Figures 6 and 7 show two syntax diagrams of one command, CDS, in HSC v1.0.2 and v2.0.0, respectively. Because of the format changes for the data set name parameter, CDS is considered a modified command. Therefore, CDS is added to *Cmd*, and *dns*, the parameter used in CDS, is added to *Factor*. According to the algorithm we explained in the previous section, CDS is considered a directly affected command. Thus, CDS is shown as one of the changed commands in Table 11, and is included in the basic regression subdomain automatically generated by *Sleuth* (refer to Table 12). Tables 11 and 12 show the domain changes detected and the regression subdomains generated by *Sleuth*. A total of 21 commands in HSC v2.0.0 are detected as changed commands compared with HSC v1.0.2 (cf. column 2 in Table 11). The parameters used by these 21 commands are added to *Factor*. All new parameters (*c*, *csectname*, *ddname*, *labelname*, etc.) and changed parameters (*acs*, *cap*, *vtam*) are added to *Param*. After constructing *Cmd*, *Param* and *Factor*, the basic and extended regression subdomains are generated based on the algorithm we introduced in the previous section.

Table 12 shows both the basic and the extended regression subdomain. The size of the basic regression subdomain for HSC v2.0.0 versus v1.0.2 is much bigger than the basic regression subdomain for HSC v2.0.1 versus v2.0.0. By contrast, the number of additional commands in the extended regression subdomain is smaller for HSC v2.0.0 versus HSC v1.0.2 as compared with the regression subdomain for HSC v2.0.1 versus HSC v2.0.0. Close study of the domain model changes revealed that the basic difference between these HSC version changes revolves around changes in the format of the parameters. The changes in CDS's syntax (Figures 6 and 7) are a typical example. These changes are spread over almost the whole domain model. Therefore, there are more direct command syntax changes for this version. Consequently, the regression subdomain for HSC 2.0.0 is larger. By contrast, there are no widely shared changes between HSC v2.0.0 and v2.0.1. Most changes involve more choices for some commands. Thus, the basic regression subdomain for HSC 2.0.1 is much smaller. However, when we consider the indirect impact of

Table 11. HSC domain model changes detected by *Sleuth*

Domain changes	HSC v2.0.0 versus v1.0.2	HSC v2.0.1 versus v2.0.0
New scripting rule	None	None
Old scripting rule	None	None
Changed scripting rule	None	None
New command	None	None
Old command	LOAD	None
Changed command	ALLOC, CAPPREF, CDS, COMMPATH, DISPLAY, DRAIN, EJECT, ENTER, JOURNAL, MNTD, MODIFY, MONITOR, OPTION, RETRY, SCRPARM, SENTER, STOPMN, TRACE, UEXIT, VIEW, WARN	ALLOC, DISPLAY, EJECT, MNTD, MOUNT, OPTION, SCRPARM, STOPMN, TRACE, WARN
New intra-Cmd rule	None	None
Old intra-Cmd rule	None	None
Changed intra-Cmd rule	None	None
New parameter	c, csectname, ddname, labelname, limit, minutes, stepname	None
Old parameter	None	None
Changed parameter	acs, cap, vtam	None

Table 12. Regression subdomains generated by *Sleuth*

Regression subdomain	HSC v2.0.0 versus v1.0.2	HSC v2.0.1 versus v2.0.0
Basic regression subdomain	ALLOC, CAPPREF, CDS, COMMPATH, DISPLAY, DRAIN, EJECT, ENTER, JOURNAL, MNTD, MODIFY, MONITOR, OPTION, RELEASE, RETRY, SCRPARM, SENTER, STOPMN, SWITCH, TRACE, UEXIT, VIEW, WARN	ALLOC, DISPLAY, EJECT, MNTD, MOUNT, OPTION, SCRPARM, STOPMN, TRACE, WARN
Additional commands in extended regression subdomain	CLEAN, DISMOUNT, MOUNT MOVE, RECOVER, SET	CAPPREF, CLEAN, COMMPATH, DISMOUNT, DRAIN, ENTER, MODIFY, MONITOR, MOVE, RECOVER, RELEASE, SENTER, SET, VIEW

changes, more commands become part of the extended regression subdomain for HSC 2.0.1. This is because parameters involved in the basic regression testing subdomain of HSC 2.0.1 are shared by many other commands. According to our algorithm, it results in a bigger difference between basic and extended regression subdomain. Both basic and extended regression subdomain generation took less than two minutes with *Sleuth*. This included the identification of domain model changes.

Table 13. Test cases generated for different regression subdomains

	HSC 2.0.0 versus 1.0.2		HSC 2.0.1 versus 2.0.0	
	Basic RTS	Extended RTS	Basic RTS	Extended RTS
@10/any	SCRPARM MODIFY DISPLAY WARN ALLOC SCRPARM CAPPREF DISPLAY RETRY EJECT	SWITCH SET SCRPARM MODIFY VIEW COMMPATH MODIFY CAPPREF COMMPATH EJECT	MNTD TRACE MODIFY TRACE DISPLAY ALLOC WARN EJECT EJECT OPTION	SWITCH SCRPARM SENTER RELEASE SCRPARM MODIFY TRACE CLEAN TRACE ENTER MNTD CAPPREF ENTER MOVE SCRPARM DRAIN DRAIN

Table 13 shows test cases automatically generated by *Sleuth* under different regression subdomains for the same test generation directive '@10/any', i.e., generate 10 commands from the currently active subdomain, observing all active rules. Note that in column 3, the command SET is generated when the extended regression subdomain is active. However, SET is not part of the basic regression subdomain. That is why this command is not generated as part of the test case in column 2. *Sleuth* may have to select more than 10 commands to fulfil a sequencing rule when generating test cases. This is the reason why column 5 of Table 13 shows more than 10 commands.

Testers used these regression subdomains for a variety of test objectives, including extensive individual command test of each command in the regression subdomain (*Sleuth* provides a utility that will generate all combinations of a command template), workload and stress tests, and error recovery tests. Yet another way to use the regression testing subdomain was to use *Sleuth's* test generation directives to systematically test specific combinations of lists of parameter values (e.g., all possible drives on all actual tape silo configurations in the physical test laboratory).

They could also reuse existing test suites in whole or in part where retestable test cases where identified by covering parts of the regression testing subdomain and conforming to current syntax. In cases where syntax changed, testers could identify retestable test cases at the script level and then regenerate command templates and fully parametrized commands from then on.

We collected data over a 12-week test cycle for one expert tester who performed regression testing between HSC 1.0.2. and HSC 2.0.0. The tester used *Sleuth* on the new release. He found over three times as many errors. Tracking post release incidents showed a 30% decline as well (Figliulo, von Mayrhauser and Karcich, 1996). He also reported cutting his test generation time in half. While this is undoubtedly a very encouraging result, more extensive assessment is desirable. Unfortunately, it is very difficult to obtain precise industrial data from testers who are busy meeting

tight release deadlines. In addition, how testers use the generation capabilities of *Sleuth*, even on the same regression subdomain, can vary greatly. While this speaks for the perceived utility of *Sleuth's* versatility, it makes any sort of controlled comparison based on *in situ* observation hard if not impossible.

Our conjecture is that it is much more than the regression testing support we provided in *Sleuth* that accounts for the success: success is also due to the versatility of the test generation environment coupled with tester capability. In this sense, providing testers with the regression testing capabilities described in this paper is only one further step in giving testers a flexible environment that helps them to put their knowledge to use and reduces tedium.

6. CONCLUSION

As more and more software is built in an evolutionary manner, regression testing is becoming more and more important. Automation is a way to reduce its costs.

We approached regression test automation in two phases: phase 1 determined what should be regression tested by building a regression subdomain. We distinguished between a basic (assume impact of changes is small) and an extended regression subdomain (assume impact of changes is large). Phase 2 takes this regression test subdomain and generates new tests for it automatically, using *Sleuth*. *Sleuth* provides support for several approaches. They include (1) retest all for whole test suite reuse, (2) retest all for whole new test suite generation, (3) use selective regression testing based on a basic regression subdomain, and (4) use selective regression testing based on an extended regression subdomain. This provides testers with the flexibility to follow a variety of different testing strategies, based on what works best for their needs (quality of software and changes). In all cases, a significant amount of automation reduces the testing effort. However, test validation is still a major issue. We are currently working on including partial test oracles into our system to reduce both test generation and validation effort.

Acknowledgements

We would like to thank the *Journal's* reviewers for their helpful suggestions in revising the paper. We also gratefully acknowledge support from Storage Technology and CASI, the Colorado Advanced Software Institute, a programme of the State of Colorado that supports technology transfer research through collaborative projects between industry and academia.

References

Agrawal, H., Horgan, J. R., Kranser, E. W. and London, S. A. (1988) 'Incremental regression testing', in *Proceedings Conference on Software Maintenance—1988*, IEEE Computer Society Press, Los Alamitos CA, pp. 348–357.

Binkley, D. (1997) 'Semantics guided regression test cost reduction', *IEEE Transactions on Software Engineering*, **23**(8), 498–516.

Chen, Y.-F., Rosenblum, D. S. and Vo, K.-P. (1994) 'TestTube: a system for selective regression testing', in *Proceedings 16th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 211–220.

Debaud, J.-M., Moopen, B. and Rugaber, S. (1994) 'Domain analysis and reverse engineering', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 326–335.

Figliulo, T., von Mayrhauser, A. and Karcich, R. (1996) 'Experiences with automated system testing and *Sleuth*', in *Proceedings IEEE Aerospace Conference*, IEEE Press, Piscataway NJ, pp. 335–349.

Harrold, M. J. and Soffa, M. L. (1988) 'An incremental approach to unit testing during maintenance', in *Proceedings Conference on Software Maintenance—1988*, IEEE Computer Society Press, Los Alamitos CA, pp. 362–367.

Hartmann, J. and Robson, D. J. (1988) 'Approaches to regression testing', in *Proceedings Conference on Software Maintenance—1988*, IEEE Computer Society Press, Los Alamitos CA, pp. 368–372.

Hsia, P., Li, X., Kung, D. C., Hsu, C. T., Li, L., Toyoshima, Y. and Chen, C. (1997) 'A technique for the selective revalidation of OO software', *Journal of Software Maintenance*, **9**(4), 217–233.

Kung, D. C., Gao, J., Hsia, P., Toyoshima, Y. and Chen, C. (1996) 'On regression testing of object-oriented programs', *Journal of Systems and Software*, **32**(1), 21–40.

Leung, H. K. N. (1995) 'Selective regression testing—assumptions and fault detecting ability', *Information and Software Technology*, **37**(10), 531–537.

Leung, H. K. N. and White, L. (1989) 'Insights into regression testing', in *Proceedings Conference on Software Maintenance—1989*, IEEE Computer Society Press, Los Alamitos CA, pp. 60–69.

Leung, H. K. N. and White, L. (1990) 'A study of integration testing and software regression at the integration level', in *Proceedings Conference on Software Maintenance—1990*, IEEE Computer Society Press, Los Alamitos CA, pp. 290–301.

Leung, H. K. N. and White, L. (1991) 'A cost model to compare regression testing strategies', in *Proceedings Conference on Software Maintenance—1991*, IEEE Computer Society Press, Los Alamitos CA, pp. 201–208.

von Mayrhauser, A. and Crawford-Hines, S. (1993) 'Automated testing support for a robot tape library', in *Proceedings Fourth International Software Reliability Engineering Conference*, IEEE Computer Society Press, Los Alamitos CA, pp. 6–14.

von Mayrhauser, A., Mraz, R. T. and Walls, J. (1994a) 'Domain based regression testing', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 26–35.

von Mayrhauser, A., Mraz, R. T., Walls, J. and Ocken, P. (1994b) 'Domain based testing: increasing test case reuse', in *Proceedings International Conference on Computer Design*, IEEE Computer Society Press, Los Alamitos CA, pp. 484–491.

von Mayrhauser, A., Walls, J. and Mraz, R. T. (1994c) 'Testing applications using domain based testing and *Sleuth*', in *Proceedings International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 206–215.

Ostrand, T. J. and Weyuker, E. J. (1988) 'Using data flow analysis for regression testing', in *Proceedings Sixth Annual Pacific Northwest Software Quality Conference*, Pacific Northwest Software Quality Conference, Portland OR, pp. 233–247.

Rosenblum, D. S. and Weyuker, E. J. (1996) 'Predicting the cost-effectiveness of regression testing strategies', in *Proceedings Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering; ACM Software Engineering Notes*, **21**(6), 118–126.

Rothermel, G. and Harrold, M. J. (1993) 'A safe, efficient algorithm for regression test selection', in *Proceedings Conference on Software Maintenance—1993*, IEEE Computer Society Press, Los Alamitos CA, pp. 358–367.

Rothermel, G. and Harrold, M. J. (1994a) 'A framework for evaluating regression test selection techniques', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 201–210.

Rothermel, G. and Harrold, M. J. (1994b) 'Selecting regression tests for object-oriented software', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 14–25.

Rothermel, G. and Harrold, M. J. (1996) 'Analyzing regression test selection techniques', *IEEE Transactions on Software Engineering*, **22**(8), 529–551.

White, L. J. (1996) 'Regression testing of GUI event interactions', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 350–358.

White, L. J. and Narayanswamy, V. (1993) 'Test Manager: a regression testing tool', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 338–347.

Authors' biographies:

Anneliese von Mayrhauser received a Dipl.-Inf. degree in Informatik (1976) from the Technical University in Karlsruhe and the A.M. (1978) and Ph.D. (1979) in Computer Science from Duke University. She is currently a Professor of Computer Science at Colorado State University and Director of the Colorado Advanced Software Institute. Dr. von Mayrhauser has published on software testing, software metrics, software maintenance, reliability and performance modelling. Email: avm@cs colostate.edu

Ning Zhang received a Bachelor of Science from Beijing University in China and a Master of Science in 1998 from Colorado State University. She is currently employed by USWEST in Denver. Email: zhangn@uswest.com

